Contents

1 BAC0 1
2 Test driven development (TDD) for DDC controls 3
3 Better commissioning thanks to automatic data logging 5
4 Table of contents 7

4.1 Getting started ................................................................. 7
  4.1.1 I know nothing about Python .................................................. 7
  4.1.2 Installing a complete distribution ............................................. 7
  4.1.3 Start using pip .................................................................. 8
  4.1.4 Check that BAC0 works .......................................................... 8
  4.1.5 Where to download the source code .......................................... 8
  4.1.6 Dependencies .................................................................... 8

4.2 How to start BAC0 ............................................................... 9
  4.2.1 Define a bacnet network ......................................................... 9
  4.2.2 Lite vs Complete .................................................................. 9
  4.2.3 Use BAC0 on a different subnect (Foreign Device) ...................... 10
  4.2.4 Discovering devices on a network ............................................ 10
  4.2.5 Time Sync ........................................................................ 12
  4.2.6 Ping devices (monitoring feature) ............................................ 12
  4.2.7 Read and write (Using the BACnet instance) .............................. 12

4.3 How to define a device and interact with points ............................. 13
  4.3.1 Define a controller ................................................................ 13
  4.3.1.1 Some caveats .................................................................. 13
  4.3.2 Look for points in controller .................................................... 14
  4.3.3 Read the value of a point ......................................................... 14
  4.3.4 Writing to Points .................................................................. 14
  4.3.4.1 Simple write .................................................................. 14
  4.3.4.2 Write to an Output (Override) ............................................ 14
  4.3.4.3 Write to an Input (simulate) ............................................... 16
  4.3.4.4 Releasing an Input simulation or Output override ................. 18
  4.3.4.5 Setting a Relinquish_Default ............................................. 18
  4.3.4.6 BACnet properties ............................................................ 18

4.4 Proprietary Objects .................................................................. 20
  4.4.1 Writing to proprietary properties .............................................. 20
  4.4.1.1 How to implement ............................................................. 20
BAC0 is a Python 3 (3.4 and later) scripting application that uses BACpypes to process BACnet messages on an IP network. This library exposes simple functions to browse the BACnet network, and read & write properties from the BACnet devices.

Python is a simple language to learn and a very powerful tool for data processing. Coupled with BACnet, it becomes a great tool for testing BACnet and interacting with BACnet controllers.

BAC0 takes its name from the default IP port assigned to BACnet/IP communications - port (47808 decimal, 0xBAC0 hexadecimal).
CHAPTER 2

Test driven development (TDD) for DDC controls

BAC0 is intended for assisting BAS (building automation system) programmers, with configuring, testing, and commissioning of BAS Controllers - often called DDC (Direct Digital Control) Controllers.

Typically BAS controllers are programmed using vendor specific tools, and vendor specific programming languages to define how they will operate. The resulting programs are the controller’s sequence of operations. Different vendors, use different methods to define these sequences - including ‘block programming’, ‘graphical programming’, and ‘text/procedural programming’.

BAC0 provides a generalized (vendor-independent) means to programmatically interact with the BAS controllers, via Python and the BACnet/IP communication protocol. BAC0 allows users to test a controller even if no sensors or outputs are connected to the controller. Thanks to the BACnet out_of_service property, it is easy to write a value to the input pin(s) so the controller believes a sensor is connected, and its operating sequence will respond accordingly. Likewise, it is possible to write a value to an output pin(s) to operate any connected equipment (often called a manual command or to override an output). In fact, BAC0 exposes a great many of a controller’s BACnet Objects and Object Properties, enabling automated interactions using Python; as a simple scripting language, a powerful testing & commissioning tool, or a general application development environment.

Using BAC0 as test tool, makes automated BAS testing quick, reliable, and repeatable. Compare this to the BAS vendor provided tools, which only allow the controllers to be programmed, and where all the testing must be done manually. Very slow. Very error-prone. Now you can write your tests and re-run them as often as you need.
Chapter 3

Better commissioning thanks to automatic data logging

As you will discover, when you define a controller in BAC0, you automatically get historical data logs for every variable in the controller. All I/O points are trended every 10 seconds (by default). Meaning you can do data analysis of the controller’s operation while you’re doing your basic sequence testing. This gives you a high-level overview of the controller’s performance while highlighting trouble areas really fast.

BAC0 is not only a good tool for testing your sequence of operations while in-the-office. It is also a really good tool to assist on-site. Use it to test controller startup, operation, and balancing in-the-field. When combined with Jupyter Notebook, you are even able to create nice looking reports right from your automation code.
4.1 Getting started

4.1.1 I know nothing about Python

First, welcome to the Python community. If you’re new to Python programming, it can be hard to know where to start. I highly recommend to start with a complete distribution. That will help you a lot as the majority of important modules will be installed for you.

If you are using Windows, it will simplify your life as some modules needs a C compiler and it can be hard sometimes to compile a module by yourself.

Some examples of complete distributions are Anaconda or Enthought Canopy. As I use Anaconda, I’ll focus on this one but you’re free to choose the one you prefer.

If you are using a RaspberryPi, have a look to miniconda or berryconda. Both can allow a complete installation of modules like bokeh and Flask. For berryconda, once it’s done, run conda install pandas. This will install pandas on your RaspberryPi without the need to compile it.

4.1.1.1 Installing a complete distribution

Begin by downloading Anaconda. Install it. Once it’s done, you’ll get access to a variety of tools like:

- Spyder (and IDE to write the code)
- Anaconda Prompt (a console configured for Python)
- Jupyter Notebook (Python in your browser)
- pip (a script allowing you to install modules)
- conda (a package manager used by Anaconda)
4.1.2 Start using pip

Open the Anaconda Prompt (a console terminal with Python configured in the path)

```
pip install BAC0
```

This simple line will look in Pypi (The Python Package Index), download and install everything you need to start using BAC0

4.1.3 Check that BAC0 works

In the terminal again, type

```
python
```

This will open a python terminal. In the terminal type

```
>> import BAC0
>> BAC0.version
```

This will show you the installed version. You’re good to go.

4.1.2 Where to download the source code

https://github.com/ChristianTremblay/BAC0/

There you’ll be able to open issues if you find bugs.

4.1.3 Dependencies

- BAC0 is based on BACpypes for all BACnet/IP communication.
  Starting at version 0.9.900, BAC0 will not strictly depend on bokeh or Flask or Pandas to work. Having them will allow to use the complete set of features (the web app with live trending features) but if you don’t have them installed, you will be able to use the ‘lite’ version of BAC0 which is sufficient to interact with BACnet devices.
- It uses Bokeh for Live trending features
- It uses Pandas for every Series and DataFrame (histories)
- It uses Flask to serve the Web app (you will need to pip install flask_bootstrap)

Normally, if you have installed Anaconda, Flask, Bokeh and Pandas will already be installed. You’ll only need to install BACpypes

```
pip install bacpypes
pip install bokeh  # or conda install bokeh if using Anaconda
```

You’re ready to begin using BAC0 !
4.2 How to start BAC0

4.2.1 Define a bacnet network

Once imported, BAC0 will rely on a ‘network’ variable that will connect to the BACnet network you want to reach. This variable will be tied to a network interface (that can be a network card or a VPN connection) and all the traffice will pass on this variable.

More than one network variable can be created but only one connection by interface is supported.

Typically, we’ll call this variable ‘bacnet’ to illustrate that it represents the network. But you can call it like you want. This variable will also be passed to some functions when you will define a device for example. As the device needs to know on which network it can be found.

When creating the connection to the network, BAC0 needs to know the ip network of the interface on which it will work. It also needs to know the subnet mask (as BACnet operations often use broadcast messages). If you don’t provide one, BAC0 will try to detect the interface for you.

**Note:** If you use ios, you will need to provide a ip manually. The script is unable to detect the subnet mask yet.

By default, if Bokeh, Pandas and Flask are installed, using the connect script will launch the complete version. But you can also use the lite version if you want something simple.

Example:

```python
import BAC0
bacnet = BAC0.connect()
# or specify the IP you want to use / bacnet = BAC0.connect(ip='192.168.1.10/24')
# by default, it will attempt an internet connection and use the network adapter
# connected to the internet.
# Specifying the network mask will allow the usage of a local broadcast address
# like 192.168.1.255 instead of the global broadcast address 255.255.255.255
# which could be blocked in some cases.
# You can also use :
# bacnet = BAC0.lite() to force the script to load only minimum features.
# Please note that if Bokeh, Pandas or Flask are not installed, using connect() will...
# in fact call the lite version.
```

4.2.1.1 Lite vs Complete

**Lite**

Use Lite if you only want to interact with some devices without using the web interface or the live trending features. On small devices like Raspberry Pi on which Numpy and Pandas are not installed, it will run without problem.

To do so, use the syntax:

```python
bacnet = BAC0.lite(ip='xxx.xxx.xxx.xxx/mask')
```

On a device without all the module sufficient to run the “complete” mode, using this syntax will also run BAC0 in “Lite” mode:

```python
bacnet = BAC0.connect()
```
Complete

Complete will launch a web server with bokeh trending features. You will be able to access the server from another computer if you want.

To do so, use the syntax:

```python
bacnet = BAC0.connect(ip='xxx.xxx.xxx.xxx/mask')
```

And log to the web server pointing your browser to http://localhost:8111

Note:
To run BAC0 in “complete” mode, you need to install supplemental packages:

- flask
- flask-bootstrap
- bokeh
- pandas (numpy)

To install bokeh, using “conda install bokeh” works really well. User will also needs to “pip install” everything else.

Note: To run BAC0 in “complete” mode using a RaspberryPi, I strongly recommend using the package berryconda. This will install Pandas, numpy, already compiled for the Pi and give you access to the “conda” tool. You’ll then be able to “conda install bokeh” and everythin will be working fine. If you try to “pip install pandas” you will face issues as the RPi will have to compile the source and it is a hard taks for a so small device. berryconda gives access to a great amount of packages already compiled for the Raspberry Pi.

4.2.1.2 Use BAC0 on a different subnet (Foreign Device)

In some situations (like using BAC0 with a VPN using TUN) your BAC0 instance will run on a different subnet than the BACnet/IP network.

BAC0 support being used as a foreign device to cover those cases.

You must register to a BBMD (BACnet Broadcast Management Device) that will organize broadcast messages so they can be sent through diferent subnet and be available for BAC0.

To do so, use the syntax:

```python
my_ip = '10.8.0.2/24'
bbmdIP = '192.168.1.2:47808'
bbmdTTL = 900
bacnet = BAC0.connect(ip='xxx.xxx.xxx.xxx/mask', bbdmAddress=bbmdIP, bbmdTTL=bbmdTTL)
```

4.2.1.3 Discovering devices on a network

BACnet protocole relies on “whois” and “iam” messages to search and find devices. Typically, those are broadcast messages that are sent to the network so every device listening will be able to answer to whois requests by a iam request.
By default, BAC0 will use “local broadcast” whois message. This mean that in some situation, you will not see by default the global network. Local broadcast will not traverse subnets and won’t propagate to MSTP network behind BACnet/IP-BACnet/MSTP router that are on the same subnet than BAC0.

This is done on purpose because using “global broadcast” by default will create a great amount of traffic on big BACnet network when all devices will send their “iam” response at the same time.

Instead, it is recommended to be careful and try to find devices on BACnet networks one at a time. For that though, you have to “already know” what is on your network. Which is not always the case. This is why BAC0 will still be able to issue global broadcast whois request if explicitly told to do so.

The recommended function to use is

```python
bacnet.discover(networks=['listofnetworks'], limits=(0,4194303), global_broadcast=False)
```

This function will trigger the whois function and get you results. It will also emit a special request named “What-si-network-number” to try to learn the network number actually in use for BAC0. As this function have been added in the protocole 2008, it may not be available on all networks.

BAC0 will store all network number found in the property named `bacnet.known_network_numbers`. User can then use this list to work with discover and find everything on the network without issuing global broadcasts. To make a discover on known networks, use

```python
bacnet.discover(networks='known')
```

Also, all found devices can be seen in the property `bacnet.discoveredDevices`. This list is filled with all the devices found when issuing whois requests.

BAC0 also provide a special functions to get a device table with details about the found devices. This function will try to read on the network for the manufacturer name, the object name, and other informations to present all the devices in a pandas dataframe. This is for presentation purposes and if you want to explore the network, I recommend using discover.

Devices dataframe

```python
bacnet.devices
```

..note:: WARNING. `bacnet.devices` may in some circumstances, be a bad choice when you want to discover devices on a network. A lot of read requests are made to look for manufacturer, object name, etc and if a lot of devices are on the network, it is recommended to use whois() and start from there.

BAC0 also support the ‘Who-Is-Router-To-Network’ request so you can ask the network and you will see the address of the router for this particular BACnet network. The request ‘Initialize-Router-Table’ will be triggered on the reception of the ‘I-Am-Router-To-Network’ answer.

Once BAC0 will know which router leads to a network, the requests for the network inside the network will be sent directly to the router as unicast messages. For example

```python
# if router for network 3 is 192.168.1.2
bacnet.whois('3:*')
# will send the request to 192.168.1.2, even if by default, a local broadcast would sent the request
# to 192.168.1.255 (typically with a subnet 255.255.255.0 or /24)
```
4.2.1.4 Time Sync

You can use BAC0 to send time synchronisation requests to the network

```python
bacnet.time_sync()
# or
bacnet.time_sync('2:5')  # <- Providing an address
```

BAC0 will not accept requests from other devices.

4.2.1.5 Ping devices (monitoring feature)

BAC0 includes a way to ping constantly the devices that have been registered. This way, when devices go offline, BAC0 will disconnect them until they come back online. This feature can be disabled if required when declaring the network

```python
bacnet = BAC0.lite(ping=False)
```

By default, the feature is activated.

When reconnecting after being disconnected, a complete rebuild of the device is done. This way, if the device have changed (a download have been done and point list changed) new points will be available. Old one will not.

.. note:: WARNING. When BAC0 disconnects a device, it will try to save the device to SQL.

4.2.1.6 Read and write (Using the BACnet instance)

BAC0 typically use the concept of controller that we’ll see later. But At its core, read and write operation will be done through the BACnet instance (`bacnet = BAC0.lite()` for example).

**Read property**

Once you know the device you need to read from, you can use

```python
bacnet.read('address object object_instance property')
```

**Read properties**

Read property multiple can also be used

```python
bacnet.readMultiple('address object object_instance property_1 property_2') #or
bacnet.readMultiple('address object object_instance all')
```

**Write to property**

To write to a single property

```python
bacnet.write('address object object_instance property value - priority')
```
Write to multiple properties

Write property multiple is also implemented. You will need to build a list for your requests.

```python
r = ['analogValue 1 presentValue 100', 'analogValue 2 presentValue 100', 'analogValue 3 presentValue 100 - 8', '@obj_142 1 @prop_1042 True']
bacnet.writeMultiple(addr='2:5', args=r, vendor_id=842)
```

.. note:: WARNING. See the section on Proprietary objects and properties for details about vendor_id and @obj_142.

4.3 How to define a device and interact with points

4.3.1 Define a controller

Once the bacnet variable is created, you can define devices.

Example:

```python
import BAC0
bacnet = BAC0.connect()
# or specify the IP you want to use / bacnet = BAC0.connect(ip='192.168.1.10/24')
# by default, it will attempt an internet connection and use the network adapter
# connected to the internet.
# Specifying the network mask will allow the usage of a local broadcast address
# like 192.168.1.255 instead of the global broadcast address 255.255.255.255
# which could be blocked in some cases.
# You can also use :
# bacnet = BAC0.lite() to force the script to load only minimum features.
# Please note that if Bokeh, Pandas or Flask are not installed, using connect() will
# in fact call the lite version.

# Get the list of devices seen on the network
bacnet.devices

# Define a controller (this one is on MSTP #3, MAC addr 4, device ID 5504)
mycontroller = BAC0.device('3:4', 5504, bacnet)

# Get the list of "registered" devices
bacnet.registered_devices
```

4.3.1.1 Some caveats

Segmentation

Some devices do not support segmentation. BAC0 will try to detect that and will not allow “read property multiple” to be used. But it is sometimes better to specify to BAC0 that the device doesn’t support segmentation.

To do so, use the parameter:

```python
my_old_device = BAC0.connect('3:4', 5504, bacnet, segmentation_supported=False)
```
Objective List

By default, BAC0 will read the object list from the controller and define every points found inside the device as points. This behaviour may not be optimal in all use cases. BAC0 allows you to provide a custom object list when creating the device.

To do so, use this syntax:

```python
# Define your own list
my_obj_list = [('file', 1),
               ('analogInput', 2),
               ('analogInput', 3),
               ('analogInput', 5),
               ('analogInput', 4),
               ('analogInput', 0),
               ('analogInput', 1)]

# Provide it as an argument
fx = BAC0.device('2:5', 5, bacnet, object_list = my_obj_list)
```

4.3.2 Look for points in controller

Example:

```python
mycontroller.points
```

4.3.3 Read the value of a point

To read a point, simply ask for it using bracket syntax:

```python
mycontroller['point_name']
```

4.3.4 Writing to Points

4.3.4.1 Simple write

If point is a value:

- analogValue (AV)
- binaryValue (BV)
- multistateValue (MV)

You can change its value with a simple assignment. BAC0 will write the value to the object’s presentValue at the default priority:

```python
mycontroller['point_name'] = 23
```

4.3.4.2 Write to an Output (Override)

If the point is an output:
4.3. How to define a device and interact with points

**Fig. 1:** Example from Delta Controls OWS Workstation

**Fig. 2:** Example from Niagara 4 station
• analogOutput (AO)
• binaryOutput (BO)
• multistateOutput (MO)

You can change its value with a simple assignment. BAC0 will write the value to the object’s **presentValue** (a.k.a override it) at priority 8 (Manual Operator):

```python
mycontroller['outputName'] = 45
```

**Fig. 3: Example from Delta Controls OWS Workstation**

### 4.3.4.3 Write to an Input (simulate)

If the point is an input:

- analogInput (AI)
- binaryOutput (BO)
- multistateOutput (MO)

You can change its value with a simple assignment, thus overriding any external value it is reading and simulating a different sensor reading. The override occurs because BAC0 sets the point’s **out_of_service** (On) and then writes to the point’s **presentValue**.

```python
mycontroller['inputName'] = <simulated value>
mycontroller['Temperature'] = 23.5 # overiding actual reading of 18.8 C
```
4.3. How to define a device and interact with points

Fig. 4: Example from Niagara 4 station

Fig. 5: Example from Delta Controls OWS Workstation
In a Niagara station, you would need to create a new point using the “out_of_service” property, then set this point to True. Then you would need to create (if not already done) a point writable to the present value property and write to it. No screenshot available.

4.3.4.4 Releasing an Input simulation or Output override

To return control of an Input or Output back to the controller, it needs to be released. Releasing a point returns it automatic control. This is done with an assignment to ‘auto’:

```python
mycontroller['pointToRelease'] = 'auto'
```

![Image of a Niagara station](image)

**Fig. 6:** Example from Delta Controls OWS Workstation

In a Niagara station, you would need to create a new point using the “out_of_service” property, then set this point to False. No screenshot available.

4.3.4.5 Setting a Relinquish_Default

When a point (with a priority array) is released of all override commands, it takes on the value of its Relinquish_Default. [BACnet clause 12.4.12] If you wish to set this default value, you may with this command:

```python
mycontroller['pointToChange'].default(<value>)
mycontroller['Output'].default(75)
```

![Image of a Niagara station](image)

**Fig. 7:** Example from Delta Controls OWS Workstation

**Fig. 8:** Example from Delta Controls OWS Workstation

4.3.4.6 BACnet properties

BAC0 defines its own “image” of a controller. All points inside a BAC0.device are Python objects with which we can interact. If you want to access native BACnet objects and properties there are functions you can use.
Read all device properties

You can retrieve the list of device properties using:

```python
device.bacnet_properties
# will return a cached version by default. If things have changed, you can refresh
device.update_bacnet_properties()
```

Often, in this list, you will see proprietary properties added by the manufacturer. They can be recognize by their name, an integer.

Read Property

You can read simple properties using

```python
prop = ('device',100,'objectName')
device.read_property(prop)
# this will return the object name
prop = ('analogInput',1,'priorityArray')
device.read_property(prop)
# this will return the priority array of AI1
```

Write to property

You can write to a property using

```python
prop = ('analogValue',1,'presentValue')
bacnet.write_property(prop,value=98,priority=7)
```
4.4 Proprietary Objects

Some manufacturers provide special variables inside their controllers in the form of proprietary objects or expand some objects with proprietary properties. BAC0 supports the creation of those objects but some work is needed on your side to register them.

In fact, you will need to know what you are looking for when dealing with proprietary objects or properties. Should you write to them or make them read only? What type should you declare?

Once you know the information, you are ready to make your implementation.

The actual BAC0 implementation allow the user to be able to read proprietary objects or proprietary properties without defining a special class. This is done using a special syntax that will inform BAC0 of the nature or the read.

Why? Bacpypes requests (in BAC0) are made sequentially using well-known property names and address. When dealing with proprietary objects or properties, names and addresses are numbers. This is somewhat hard to detect if the request contains an error, is malformed or contains a proprietary thing in it. The new syntax will tell BAC0 that we need to read a proprietary object or property.

If you need to read an object named “142”, you will tell BAC0 to read `@obj_142` If you need to read a property named 1032, you will tell BAC0 to read `@prop_1032`

This way, you could build a request this way:

```python
bacnet.read('2:5 @obj_142 1 @prop_1032') # or bacnet.readMultiple('2:5 @obj_142 1 objectName @prop_1032')
```

4.4.1 Writing to proprietary properties

If you need to write to the property, things are a little more complicated. For example, JCI TEC3000 have a variable that needs to be written to so the thermostat know that the supervisor is active, a condition to use network schedule (if not, switch to internal schedule).

If you try this:

```python
bacnet.write('2000:10 device 5010 3653 True')
```

You'll get:

```
TypeError: issubclass() arg 1 must be a class
```

This is because BAC0 doesn't know how to encode the value to write. You will need to define a class, register it so BAC0 knows how to encode the value and most importantly, you will need to provide the `vendor_id` to the write function so BAC0 will know which class to use. Because 2 different vendors could potentially use the same “number” for a proprietary object or property with different type.

4.4.1.1 How to implement

BAC0 will allow dynamic creation of the classes needed to read and write to those special variables. To do so, a special dictionary need to be declared in this form ::

```python
defineProprietary(name: str, vendor_id: int, objectType: str, bacpypes_type: type, properties: dict) -> type:
    obj = type(name, (Object,), {
        "name": name,
        "vendor_id": vendor_id,
        "objectType": objectType,
        "bacpypes_type": bacpypes_type,
        "properties": {
            "NameOfProprietaryProp": {"obj_id": 1110, "primitive": Boolean, "mutable": True},
        }
    })
    return obj
```

(continues on next page)
Once the dictionary is completed, you need to call the special function `create_proprietaryobject`. This function will dynamically create the class and register it with bacpypes so you will be able to read and write to the object.

To access the information (for now), you will use this syntax:

```python
# Suppose an MSTP controller at address 2:5, device instance 5003
# Vendor being Servisys (ID = 842)
# Proprietary property added to the device object with object ID 1234
bacnet.read('2:5 device 5003 1234', vendor_id=842)
```

If you want to look at the object registration, you can use this

```python
from bacpypes.object import registered_object_types
registered_object_types
```

It is a dictionary containing all the registered type in use. As you can see, the majority of the registration use vendor_id 0 which is the default. But if you register something for another vendor_id, you will see a new dictionary entry. Using the special `bacnet.read` argument “vendor_id” will then inform bacpypes that we want to use the special object definition for this particular vendor.

**Note:** BAC0 will automatically register known proprietary classes at startup. See BAC0.core.proprietary_objects for details.

### 4.4.1.2 Proprietary objects

Proprietary object can be accessed using

```python
# Let say device '2:5' have object (140,1)
bacnet.read('2:5 140 1 objectName')
```

As they are proprietary objects, you will have to know what you are looking for. Typically, the properties `objectName`, `objectIdentifier`, will be available. But you will often see proprietary properties attached to those objects. See next section.

To read all properties from an object, if implemented, one can use

```python
bacnet.readMultiple('2:5 140 1 all')
```

BAC0 will do its best to give you a complete list.
Note: Please note that arrays under proprietary objects are not implemented yet. Also, context tags objects are not detected automatically. You will need to build the object class to interact with those objects. See next section.

4.4.1.3 Proprietary Property

One common case I’m aware of is the addition of proprietary properties to the DeviceObject of a device. Those properties may, for example, give the CPU rate or memory usage of the controllers. On the TEC3000 (JCI), there is a “SupervisorOnline” property needed to be written to, allowing the BAS schedule to work.

To declare those properties, we need to extend the base object (the DeviceObject in this case) pointing this declaration to the vendor ID so bacpypes will know where to look.

The following code is part of BAC0.core.proprietary_objects.jci and define proprietary properties added to the device object for JCI devices. Note that as there are multiple proprietary properties, we need to declare them all in the same new class (the example presents 2 new properties).

```python
# Proprietary Objects and their attributes
#
JCIDeviceObject = {
    "name": "JCI_DeviceObject",
    "vendor_id": 5,
    "objectType": "device",
    "bacpypes_type": DeviceObject,
    "properties": {
        "SupervisorOnline": {"obj_id": 3653, "primitive": Boolean, "mutable": True},
        "Model": {"obj_id": 1320, "primitive": CharacterString, "mutable": False},
    },
}
```

This will allow us to interact with them after registration

```python
from BAC0.core.proprietary_objects.jci import JCIDeviceObject
from BAC0.core.proprietary_objects.object import create_proprietaryobject
create_proprietaryobject(**JCIDeviceObject)

# Read model of TEC
bacnet.read('2:5 device 5005 1320', vendor_id=5)
# Write to supervisor Online
bacnet.write('2:5 device 5005 3653 true', vendor_id=5)
```

Note: In future version it will be able to define special device and attach some proprietary objects to them so tec[‘SupOnline’] would work...

4.4.2 Vendor Context for Read and Write

In BAC0.device, the vendor_id context will be provided to the stack automatically. This mean that if a device is created and there is a extended implementation of an object (JCIDeviceObject for example) BAC0 will recognize the proprietary object by default, without having the need to explicitly define the vendor_id in the request.
instance_number = 1000
prop_id = 1320
device.read_property(('device', instance_number, prop_id))

will work.
Also, proprietary objects and properties classes are defined at startup so it is not necessary to explicitly register them.

4.4.3 Can proprietary objects be added to a BAC0.device points

Actually not, because of the way “points” are defined in BAC0. If you look at BAC0.core.devices.Points.Point you will see that the notion of point is oriented differently than a BACnet object. Properties are a set of informations useful for BAC0 itself but are not “strictly” BACnet properties. The value of a point will always be the presentValue of the BACnet object. In the context of proprietary objects, this can’t fit.

There are no “standard” way to create a proprietary object. Beside the fact that objectName, objectType and objectIdentifier must be provided, everything else is custom.

For this reason, proprietary objects must be dealt outside of the scope of a device, especially in the context of writing to them.

4.4.4 How to implement readMultiple with proprietary objects and properties

It is possible to create read property multiple requests with them, using the syntax @obj_ and @prop_. So for now, you will be able to create a request yourself for one device at a time by chaining properties you want to read:

    bacnet.readMultiple('2000:31 device 5012 @prop_3653 analogInput 1106 presentValue units')

4.4.5 How to find proprietary objects and properties

In BAC0, for a device or a point, you can use:

    device.bacnet_properties # or point.bacnet_properties

This will list all properties in the object. (equivalent of bacnet.readMultiple('addr object id all'))

4.5 Histories in BAC0

Histories in BAC0 were first introduced as the result of every reading the software made on each points, to keep trace of what was going on. It is different from the BACnet TrendLog object that may be configured in a device to keep history records in the memory of the controller.

Everytime a value is read in BAC0, the value will be stored in memory as what will be called from here: history.

TrendLog will also being accessible but for now, let’s focus on BAC0’s histories.

BAC0 uses the Python Data Analysis library pandas [http://pandas.pydata.org/] to maintain histories of point values over time. All points are saved by BAC0 in a pandas Series every 10 seconds (by default). This means you will automatically have historical data from the moment you connect to a BACnet device.

Access the contents of a point’s history is very simple:

    controller['pointName'].history

Example
controller['Temperature'].history

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-03-30</td>
<td>12:50:46</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:50:56</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:51:07</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:51:17</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:51:28</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:51:38</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:51:48</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:51:59</td>
<td>19.632507</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:52:09</td>
<td>19.536366</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:52:20</td>
<td>19.536366</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:52:30</td>
<td>19.536366</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:52:40</td>
<td>19.536366</td>
</tr>
<tr>
<td>2017-03-30</td>
<td>12:52:50</td>
<td>19.536366</td>
</tr>
</tbody>
</table>

dtype: float64

**Note:** pandas is an extensive data analysis tool, with a vast array of data manipulation operators. Exploring these is beyond the scope of this documentation. Instead we refer you to this cheat sheet [https://github.com/pandas-dev/pandas/blob/master/doc/cheatsheet/Pandas_Cheat_Sheet.pdf] and the pandas website [http://pandas.pydata.org/].

### 4.5.1 History Size

By default, BAC0 doesn’t provide a history_size per points (number of records). But it could be useful in certain cases when the script will run for a long period of time and you want to keep control over memory.

```python
dev = BAC0.device('2:4', 4, bacnet, history_size=2)
# or after...
device.update_history_size(100)
# or just on one point:
device['point'].properties.history_size = 30
```

### 4.5.2 Resampling data

One common task associated with point histories is preparing it for use with other tools. This usually involves (as a first step) changing the frequency of the data samples - called resampling in pandas terminology.

Since the point histories are standard pandas data structures (DataFrames, and Series), you can manipulate the data with pandas operators, as follows:

```python
# code snippet showing use of pandas operations on a BAC0 point history.
# Resample (consider the mean over a period of 1 min)
tempPieces = {
    '102_ZN-T' : local102['ZN-T'].history.resample('1min'),
    '102_ZN-SP' : local102['ZN-SP'].history.resample('1min'),
    '104_ZN-T' : local104['ZN-T'].history.resample('1min'),
    '104_ZN-SP' : local104['ZN-SP'].history.resample('1min'),
    '105_ZN-T' : local105['ZN-T'].history.resample('1min'),
    '105_ZN-SP' : local105['ZN-SP'].history.resample('1min'),
    '106_ZN-T' : local106['ZN-T'].history.resample('1min'),
    '106_ZN-SP' : local106['ZN-SP'].history.resample('1min'),
    '109_ZN-T' : local109['ZN-T'].history.resample('1min'),
    '109_ZN-SP' : local109['ZN-SP'].history.resample('1min'),
```

(continues on next page)
'110_ZN-T' : local110['ZN-T'].history.resample('1min'),
'110_ZN-SP' : local110['ZN-SP'].history.resample('1min'),
}

# Remove any NaN values
temp_pieces = pd.DataFrame(tempPieces).fillna(method = 'ffill').fillna(method = 'bfill')

# Create a new column in the DataFrame which is the error between setpoint and temperature
temp_pieces['Erreur_102'] = temp_pieces['102_ZN-T'] - temp_pieces['102_ZN-SP']
temp_pieces['Erreur_104'] = temp_pieces['104_ZN-T'] - temp_pieces['104_ZN-SP']
temp_pieces['Erreur_105'] = temp_pieces['105_ZN-T'] - temp_pieces['105_ZN-SP']
temp_pieces['Erreur_106'] = temp_pieces['106_ZN-T'] - temp_pieces['106_ZN-SP']
temp_pieces['Erreur_109'] = temp_pieces['109_ZN-T'] - temp_pieces['109_ZN-SP']
temp_pieces['Erreur_110'] = temp_pieces['110_ZN-T'] - temp_pieces['110_ZN-SP']

# Create a new dataframe from results and show some statistics
temp_erreurs = temp_pieces[['Erreur_102', 'Erreur_104', 'Erreur_105', 'Erreur_106',
                            'Erreur_109', 'Erreur_110']]
temp_erreurs.describe()

### 4.6 TrendLog

BACnet TrendLog is an object that a controller may implement. Once configured, it’ll keep in memory a certain number of records for one particular point. Often though, as the controller memory may be limited, we’ll have access to a limited number of records and the interval between each record may have been configured long enough to optimize the time frame coverage of the records. For example, taking records every 10 minutes instead of every 2 minutes will give 5 times longer time interval. Using a total of 2000 number of records, it will represent almost 14 days (10 min) vs less than 3 days (2 min).

BAC0 supports the reading of TrendLog objects and will convert the records to pandas Series if possible. This will allow to use pandas syntax over histories and make analysis easier for the user.

TrendLog objects have also been made compatible with the format required to be added as Bokeh Chart in the web interface:

```python
# Manually create a TrendLog
import BAC0
bacnet = BAC0.connect()
device = BAC0.device('2:5',5,bacnet)

trend = BAC0.TrendLog(1,device)

trend.history

# Adding this object to live trends
trend.chart()
```
4.7 Trends

Trending is a nice feature when you want to see how a points value changed over time. This is only possible using matplotlib directly in Jupyter. And also in the Web Interface using Bokeh [http://bokeh.pydata.org/en/latest/] which brings a complete set of wonderful features for visualizing point histories (a.k.a. trends). The best feature of all - the ability to see Live Trends of your data as it occurs.

4.7.1 Matplotlib

Matplotlib is a well known data plotting library for Python. As BAC0’s historical point data are pandas Series and DataFrames, it’s possible to use Matplotlib with BAC0. i.e. Showing a chart using matplotlib:

```python
%matplotlib notebook
# or matplotlib inline for a basic interface
controller['nvoAll'].history.plot()
```

![Figure 1](image)

4.7.2 Seaborn

Seaborn is a library built over Matplotlib that extends the possibilities of creating statistical trends of your data. I strongly suggest you have a look to this library.
4.7.3 Bokeh

Bokeh is a Python interactive visualization library targeting modern web browsers for presentation. Its goal is to provide elegant, concise graphics, with high-performance interactivity over very large or streaming datasets. Bokeh can help anyone who would like to quickly create interactive plots, dashboards, and data applications.

Note: BAC0 trending features use Bokeh when running in “complete” mode. This requires the user to have some libraries installed:

- bokeh
- flask
- flask-bootstrap
- pandas
- numpy

Note: Running in “complete” mode may be hard to accomplish if you are running BAC0 on a Raspberry Pi. If doing so, I strongly recommend using the package berryconda which will install everything you need on the RPi to use Pandas, numpy… already compiled for the RPi.

A simple call for “conda install bokeh” will install the package.

4.7.4 A web interface

To simplify the usage of the live trending feature, BAC0 implements a Web Server (running with Flask). Connect to http://localhost:8111 and you will get access to a Dashboard and the Trends page.

Internally, BAC0 will run two servers (flask and a bokeh server) that will handle the connection to the web interface and provide the web page with a live trend of the charts that have been sent to the interface.

4.7.5 Add/Remove plots to Bokeh

At first, the web page will be empty and no trend will appear. The user needs to specify which points must be trended. Points to trend are added to a list monitored by the “network” object. This will allow to add trends coming from multiple controllers easily.

```
#each point can be added
ccontroller['nvoAII'].chart()

#for we can add them using the “network” object
bacnet.add_chart(controller['nvoAII'])

# TrendLog object can also be added
trendlog_object.chart()
```

The list of trended points can be retrieve

```
bacnet.trends
#will give a list of all points added
```

To remove points
for the point directly
controller['nvoAI1'].chart(remove=True)
bacnet.remove_chart(controller['nvoAI1'])

4.7.6 Bokeh Features

Bokeh has an extensive set of features. Exploring them is beyond the scope of this documentation. Instead you may
discover them yourself at [http://www.bokehplots.com]. A couple of its features are highlighted below.

Hover tool:
And a lot of other options like pan, box zoom, mouse wheel zoom, save, etc…:

By default, x-axis will be a timeseries and will be linked between trends. So if you span one, or zoom one, the other plots will follow, giving you the exact same x-axis for every plots.

4.7.7 Bokeh Demo

Here is a working demo of Bokeh. It’s taken from a real life test. You can use all the features (zoom, pan, etc.) Please note that the hover suffers from a little bug in this “saved” version of the trends… Working to solve this.
4.8 Saving your data

When doing tests, it can be useful to go back in time and see what happened before. BAC0 allows you to save your progress (historical data) to a file that you’ll be able to re-open in your device later.

Use

```python
controller.save()
```

and voila! Two files are created. One (an SQLite file) contains all the histories, and one binary file containing all the details and properties of the device so the details can be rebuilt when needed.

By default, the ‘object name’ of the device is used as the filename. But you can specify a name

```python
controller.save(db='new_name')
```

4.8.1 Offline mode

As already explained, a device in BAC0, if not connected (or cannot be reached) will be created as an offline device. If a database exists for this device, it will automatically loaded and all the points and histories will be available just as if if you were actually connected to the network.

You can also force a connection to use an existing database if needed. Provide connect function with the desired database’s name:

```python
controller.connect(db='db_name')
```

Please note: this feature is experimental.

4.8.2 Saving Data to Excel

Though the use of the Python module xlwings [https://www.xlwings.org/], it’s possible to export all the data of a controller into an Excel Workbook.

Example

```python
controller.to_excel()
```

4.9 Web Interface

4.9.1 Flask App

BAC0 when used in “Complete” mode will start a Web app that can be reached with a browser. The app will present the bokeh server feature for live trending.

More documentation will come in the future as this feature is under development.

4.10 Demo in a Jupyter Notebook

When installed, module can be used to script communication with bacnet device. Jupyter Notebooks are an excellent way to test it. Here is an example.
4.11 Testing and simulating with BAC0

BAC0 is a powerful BAS test tool. With it you can easily build tests scripts, and by using its `assert` syntax, you can make your DDC code stronger.

### 4.11.1 Using Assert and other commands

Let’s say your BAC controller *sequence of operation* is really simple. Something like this:

<table>
<thead>
<tr>
<th>System stopped:</th>
</tr>
</thead>
<tbody>
<tr>
<td>When system <em>is</em> stopped, fan must be off, dampers must be closed, heater cannot operate.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System started:</th>
</tr>
</thead>
<tbody>
<tr>
<td>When system starts, fan command will be on. Dampers will <em>open</em> to minimum position. If fan status turns on, heating sequence will start.</td>
</tr>
</tbody>
</table>

And so on...

### 4.11.2 How would I test that?

Assuming:

- Controller is defined and its variable name is `mycontroller`
- fan command = `SF-C`
- Fan Status = `SF-S`
- Dampers command = `MAD-O`
- Heater = `RH-O`
- Occupancy command = `OCC-SCHEDULE`

**System Stopped Test Code:**

```python
mycontroller['OCC-SCHEDULE'] = 'Unoccupied'
time.sleep(10)
assert mycontroller['SF-C'] == False
assert mycontroller['MAD-O'] == 0
assert mycontroller['RH-O'] == 0

# Simulate fan status as SF-C is Off
mycontroller['SF-S'] = 'Off'
```

**System Started Test Code:**

```python
mycontroller['OCC-SCHEDULE'] = 'Occupied'
time.sleep(10)
assert mycontroller['SF-C'] == 'On'
# Give status
mycontroller['SF-S'] = 'On'
time.sleep(15)
assert mycontroller['MAD-O'] == mycontroller['MADMIN-POS']
```
And so on. . .
You can define any test you want. As complex as you want. You will use more precise conditions instead of a simple
time.sleep() function - most likely you will read a point value that tells you when the actual mode is active.
You can then add tests for the various temperature ranges; and build functions to simulate discharge air temperature
depending on the heating or cooling stages… it’s all up to you!

4.12 Using tasks to automate simulation

4.12.1 Polling

Let’s say you want to poll a point every 5 seconds to see how the point reacted.:  
```python
mycontroller['point_name'].poll(delay=5)
```

Note: by default, polling is enabled on all points at a 10 second frequency. But you could define a controller
without polling and do specific point polling.: 
```python
mycontroller = BAC0.device('2:5',5,bacnet,poll=0) mycontroller['point_name'].poll(delay=5)
```

4.12.2 Match

Let’s say you want to automatically match the status of a point with it’s command to find times when it is reacting to
conditions other than what you expected.: 
```python
mycontroller['status'].match(mycontroller['command'])
```

4.12.3 Custom function

You could also define a complex function, and send that to the controller. This way, you’ll be able to continue using
all synchronous functions of Jupyter Notebook for example. (technically, a large function will block any inputs until
it’s finished)

Note: THIS IS A WORK IN PROGRESS

Example

```python
import time
def test_Vernier():
    for each in range(0,101):
        controller['Vernier Sim'] = each
        print('Sending : %2f' % each)
        time.sleep(30)
controller.do(test_Vernier)
```

This function updates the variable named “Vernier Sim” each 30 seconds; incrementing by 1 percent. This will take a
really long time to finish. So instead, use the “do” method, and the function will be run is a separate thread so you are
free to continue working on the device, while the function commands the controller’s point.
4.13 Using Pytest

Pytest [https://docs.pytest.org/en/latest/] is a “a mature full-featured Python testing tool”. It allows the creation of test files that can be called by a command line script, and run automatically while you work on something else.

For more details, please refer Pytest’s documentation.

4.13.1 Some basic stuff before we begin

Pytest is a very simple testing tool. While, the default unit test tool for python is unittest (which is more formal and has more features); unittest can easily become too much for the needs of testing DDC controllers.

Pytest uses only simple the assert command, and locally defined functions. It also allows the usage of “fixtures” which are little snippets of code that prepare things prior to the test (setUp), then finalize things when the test is over (tearDown).

The following example uses fixtures to establish the BACnet connection prior to the test, and then saves the controller histories and closes the connection after the tests are done.

4.13.1.1 Example

Code

```python
import BAC0
import time
import pytest

# Make a fixture to handle connection and close it when it's over
@ pytest.fixture(scope='module')
def bacnet_network(request):
    print("Let's go!")
    bacnet = BAC0.connect()
    controller = BAC0.device('2:5', 5, bacnet)

    def terminate():
        controller.save()
        bacnet.disconnect()
        print('It's over')
        request.addfinalizer(terminate)
    return controller

def test_input1_is_greater_than_zero(bacnet_network):
    assert controller['nvoAI1'] > 0

def test_input2_equals_fifty(bacnet_network):
    assert controller['nvoAI2'] > 0

def test_stop_fan_and_check_status_is_off(bacnet_network):
    controller['SF-C'] = False
    time.sleep(2)
    assert controller['SF-S'] == False

def test_start_fan_and_check_status_is_on(controller):
    controller['SF-C'] = True
    time.sleep(2)
    assert controller['SF-S'] == True
```

4.13. Using Pytest
BAC0 Documentation

Success result

If you name the file: test_mytest.py, you can just run

```
py.test -v -s
```

Pytest will look for the test files, find them and run them. Or you can define the exact file you want to run

```
py.test mytestfile.py -v -s
```

Here’s what it looks like

```
=============== test session starts ===============
platform win32 -- Python 3.4.4, pytest-2.8.5, py-1.4.31, pluggy-0.3.1 -- C:\Users\ctremblay.SERVISYS\AppData\Local\Continuum\Anaconda3\python.exe
cachedir: .cache
rootdir: c:\Programmes\Github\BAC0, inifile:
plugins: bdd-2.16.1, cov-2.2.1, pep8-1.0.6
collected 2 items

test_example.py::test_input1_is_greater_than_zero Let's go!
Starting app...
App started
Starting Bokeh Serve
Click here to open Live Trending Web Page
http://localhost:5006/?bokeh-session-id=um2kEfhn97a1VQOr3GRu5t07hvQItkruMVUUDpshS8Ha
Changing device state to <class 'BAC0.core.devices.Device.DeviceDisconnected'>
Changing device state to <class 'BAC0.core.devices.Device.RPMDeviceConnected'>
Found FX14 0005... building points list
Failed running bokeh.bat serve
Bokeh server already running
Ready!
Polling started, every values read each 10 seconds
PASSED
test_example.py::test_input2_equals_fifty PASSEDFile exists, appending data...
FX14 0005 saved to disk
Stopping app
App stopped
It's over

=============== 2 passed in 27.94 seconds ===============
```

Failure result

Here’s what a test failure looks like:

```
=============== test session starts ===============
platform win32 -- Python 3.4.4, pytest-2.8.5, py-1.4.31, pluggy-0.3.1 -- C:\Users\ctremblay.SERVISYS\AppData\Local\Continuum\Anaconda3\python.exe
cachedir: .cache
rootdir: c:\Programmes\Github\BAC0, inifile:
plugins: bdd-2.16.1, cov-2.2.1, pep8-1.0.6
```

(continues on next page)
collected 2 items

pytest_example.py::test_input1_is_greater_than_zero Let's go!
Starting app...
App started
Starting Bokeh Serve
Click here to open Live Trending Web Page
http://localhost:5006/?bokeh-session-id=TKgDiRoCkut2iobSFR1WGA2nhJ1PCTXU3ZTWL3cCnxRI
Changing device state to <class 'BAC0.core.devices.Device.DeviceDisconnected'>
Changing device state to <class 'BAC0.core.devices.Device.RPMDeviceConnected'>
Found FX14 0005... building points list
Failed running bokeh.bat serve
Bokeh server already running
Ready!
Polling started, every values read each 10 seconds
PASSED
pytest_example.py::test_input2_equals_fifty FAILED
File exists, appending data...
FX14 0005 saved to disk
Stopping app
App stopped
It's over

================================== FAILURES ===================================
__________________________ test_input2_equals_fifty ___________________________
controller = FX14 0005 / Connected

def test_input2_equals_fifty(controller):
    >   assert controller['nvoAI2'] > 1000
E   assert nvoAI2 : 20.58 degreesCelsius > 1000

pytest_example.py:30: AssertionError

Note: I modified the test to generate a failure - nvoAI2 cannot exceed 1000.

4.13.2 Conclusion

Using Pytest is a really good way to generate test files that can be reused and modified depending on different use cases. It’s a good way to run multiple tests at once. It provides concise reports of every failure and tells you when your tests succeed.

4.14 Logging and debugging

All interactions with the user in the console is made using logging and an handler. Depending on the user desire, the level can be adjusted to limit or extend the verbosity of the app.

It is not recommended to set the stdout to logging.DEBUG level as it may fill the shell with messages and make it very hard to enter commands. Typically, ‘debug’ is sent to the file (see below).
By default, stderr is set to logging.CRITICAL and is not used; stdout is set to logging.INFO; file is set to logging.WARNING. The goal behind to not fill the file if it is not explicitly wanted.

### 4.14.1 Level

You can change the logging level using

```python
import BAC0
BAC0.log_level(level)
# level being 'debug, info, warning, error'
# or
BAC0.log_level(log_file=logging.DEBUG, stdout=logging.INFO, stderr=logging.CRITICAL)
```

### 4.14.2 File

A log file will be created under your user folder (~)/.BAC0. It will contain warnings by default until you change the level.

Extract from the log file (with INFO level entries)

<table>
<thead>
<tr>
<th>Time</th>
<th>Level</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>Starting app...</td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>BAC0 started</td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>Changing device state to <code>&lt;class 'BAC0.core.devices.Device.DeviceDisconnected'&gt;</code></td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td># Read property</td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td># Read property</td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>'segmentedTransmit' <code>&lt;class 'bacpypes.basetypes.Segmentation'&gt;</code></td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>Changing device state to <code>&lt;class 'BAC0.core.devices.RPMDeviceConnected'&gt;</code></td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>'analogValue' : 15</td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>propertyIdentifier 0 propertyArrayIndex value datatype</td>
</tr>
<tr>
<td>2018-04-08</td>
<td>INFO</td>
<td>'nciPIDTPRdCTI' <code>&lt;class 'bacpypes.primitivedata.CharacterString'&gt;</code></td>
</tr>
</tbody>
</table>

(continues on next page)
4.14. Logging and debugging

2018-04-08 21:47:30,745 - INFO | 'presentValue' None 800.0 <class 'bacpypes.primitivedata.Real'>
2018-04-08 21:47:30,745 - INFO | 'units' None 'seconds' <class 'bacpypes.basetypes.EngineeringUnits'>
2018-04-08 21:47:30,746 - INFO | 'description' None 'nciPIDTPdCTI' <class 'bacpypes.primitivedata.CharacterString'>
2018-04-10 23:18:26,184 - DEBUG | BAC0.core.app.ScriptApplication ForeignDeviceApplication ('do_IAmRequest %r', <bacpypes.apdu.IAmRequest(0) instance at 0x9064c88>)
CHAPTER 5

Developer documentation
CHAPTER 6

Index and search tool

• genindex
• modindex
• search